COAM

Center for Ocean & Atmospheric Modeling

Building 1103
Stennis Space Center
Mississippi 39529

Phone 601-688-5737
FAX 601-688-7072

GLENDA SOFTWARE DESIGN

Benjamin R. Seyfarth

Jerry L. Bickham

Germana Peggion

The University of Southern Mississippi

TR-3/95
February 1995

Executive Summary

Computer capacity was, and marginally still is, a limiting factor for the development of basin-wide, eddy-resolving ocean circulation models. For example, the North Atlantic Ocean configuration should extend from the American (85W) to European/African (0E) coasts, from 20S (to include the Tropical Convergence Zone) to the Fram Straits (75N). A $1/4$ degree resolution (the average grid spacing for a minimum parameterization of mesoscale variability) requires a horizontal mesh of about (344 x 384) points. To reproduce the vertical structure of the ocean dynamics and include the interactions of the several water masses that are present in the basin, the vertical resolution should include at least 15-20 collocation points. This implies a grid mesh of about 2×10^6 points, with a heavy burden on computer memory and computation requirements. Parallel processing may alleviate this problem. In view of basin-wide applications, it makes sense to divide the data matrix for the basin into subdomains and to allocate one CPU for each subdivision of the original matrix. With this allocation, each processor updates its matrix and then propagates the boundary values to its neighbors.

In order to provide an adequate representation of the mesoscale features, an alternative is to develop a modeling system that connects a large-scale, basin-wide model with high-resolution, regional models concentrated in selected areas of the domain. However, a few unsolved questions are associated with such an approach. First of all, it is necessary to implement nesting algorithms that ensure a correct transfer of energy between the coarse and fine grids. Moreover, it is necessary to develop appropriate computational techniques that keep a continuous exchange of information as the computations proceed. Again, parallel processing can provide such a tool: two distinct networks of processors perform the coarse and fine grid computations, separately; the updated interface variables are communicated to a central server that applies the nesting algorithms and transmits the new data back to the networks.

This document introduces a new parallel processing software system and evaluates and discusses its feasibility to ocean circulation modeling. Applications will be presented in a following document.

Abstract

Glenda is an environment for parallel processing which is modeled after the Linda language and utilizes the PVM software system to provide underlying communications. The resulting software maintains the friendly parallel programming, typical of Linda, and the PVM efficiency in message-passing operations.

This document describes the functions, data structure, and algorithms of the Glenda software architecture.

1 Introduction

Glenda is a parallel programming system modeled after the Linda system which is a easily-learned parallel programming environment (Carriero Gelernter, 1990). Glenda is implemented using the PVM (Parallel Virtual Machine) for communications.

PVM, developed by Oak Ridge National Laboratory (Beguelin et al., 1991), is a software system that allows the creation of and access to a concurrent computing system made from networks of loosely-coupled processing elements. The hardware collected into a user machine may be single-processor workstations, vector machines, parallel supercomputers, or any mixture of the above.

To be as general and flexible as possible, PVM is based on a parallel message-passing model. That is, the programmer must pack each item of a message into a message buffer prior to sending it and, similarly, unpack the message components upon receiving a message.

The major attributes of PVM are:

- Can use a workstation network and/or a multi-CPU system
- Architectures can be mixed on a virtual machine
- Most popular architectures are supported
- Asynchronous message passing model
- Messages consist of multiple components of eight different data types
- Messages are accessed by an integer message type
- Supports barriers and signals
- Supports Fortran and C
- Consists of 53 functions for C and 36 for Fortran

Linda, a registered trademark of Scientific Computing Associates, is a process control model which is based on a global storage system called tuple space. Processes may in/output tuples of various lengths into the tuple space, using pattern matching on tuple contents. This can be applied to model both passing and broadcasting directed messages.

Linda tuples are manipulated using six basic operations, each of which either produces or consumes a tuple, viz.,:

- **out** produces a tuple
- **in** consumes a tuple
- **inp** consumes a tuple if available
- **rd** reads a tuple
- **rdp** reads a tuple if available
- **eval** executes a function in a subprocess producing a tuple

When using the Linda model and agenda parallelism, worker processes are all equally capable and retrieve tasks from an agenda until all work is done. One master process starts worker processes using the `eval` function. The master then sends tasks using some agenda. Each worker program consists of a loop which retrieves tasks from the agenda and sends results back to the master process. When all the data have been received, the master process sends tasks with with recognizable illegal requests termed *poison pills* to the worker processes to terminate their executions.

Although the six Linda operations are adequate for performing parallel programming and are simpler to use than the analogous PVM functions, there are some significant problems to overcome for making Linda as efficient as PVM in a message-passing environment. The greatest difficulty is to avoid excessive communications.

Glenda was developed to provide the portability and efficiency of PVM with the ease-of-use of Linda (Seyfarth et al., 1994). Glenda's primary goals are:

- Preserve Linda's global tuple space model
- Reduce the number of functions required by PVM
- Maintain PVM's message passing efficiency whenever required
- Maintain PVM's portability
- Present Linda as an integral part of the host language.

Glenda applies the PVM system to perform communications and manages the global tuple space in a manner similar to that used with the Linda language. If the global tuple space is assigned to a single processor, applications might possibly experience excessive network traffic. To avoid the problem, operations have been implemented that provide direct tuple operations among the Glenda processes.

2 Glenda Primary Functions

2.1 Glenda Task Control Functions

There are three operations that control the task activities:

- **tid = gl_mytid()**
  The operation starts Glenda's activities and returns an integer which identifies the calling task in the following Glenda and PVM function calls. A process must call `gl_mytid` before any other Glenda operations.
- **tid = gl_spawn(name[,host])**
  A new task is started by a call to `gl_spawn`. The `name` parameter is the name of the executable file for the new task. The `host` parameter allows the optional designation of a specific computer to execute the program. The return value is a PVM task id. The newly-spawned task must call `gl_mytid` before executing any other Glenda functions.

- **gl_exit**
  The purpose is to perform any required cleanup before the program exits and to inform the tuple server that the task has been completed. In turn, `gl_exit` calls `pvm_exit` to inform PVM also of the task completion. However, `gl_exit` does not call `exit` and, consequently, the program continues execution.

2.2 Glenda Global Tuple Operations

There are five Glenda operations which manipulate tuples in the global tuple space. These are named and modeled after the related Linda operations, viz.:

- **gl_out(name,...)** places a tuple into the the global tuple space
- **gl_in(name,...)** gets a tuple from the global tuple space
- **gl_inp(name,...)** is a predicate version of `gl_in`. It returns 1 if a matching tuple exists and it gets the data for the tuple. It returns 0 if no matching tuple exists.
- **gl_rd(name,...)** is similar to `gl_in`. The difference is that it retrieves a copy of the matching tuple data without removing the tuple from the global tuple space.
- **gl_rdp(name,...)** is the predicate version of `gl_rd`.

All of these operations syntactically resemble C function calls with a variable number of parameters. However, there are a few differences. In the Glenda language, the first parameter for the calls is always the name of the tuple. This is either a C string constant or a C array with a `NULL`-terminated string. The name is used by the tuple server to rapidly identify the tuples. The remaining parameters are components of a tuple and may be C constants, scalar variables or arrays of the basic C data types. In read and input operations, a parameter may be preceded by a question mark, indicating that the following variable has to be replaced by the value of the matching tuple. If the variable is not preceded by a question mark, the operation must find a tuple with the same value of the parameter.

Array components of a tuple may be followed by two optional fields that define the length and stride of the array. When specified, the array length value is separated from the array name by a colon. The specification is optional if the length is known to the C-Glenda preprocessor. In `gl_out`, the length specifies how many array elements are to be copied into the tuple. For input and read operations, the length field is an integer variable indicating the length of the array to be received.

After the array length, the array stride parameter may be specified. It represents the increment for copying the array elements into the tuple. The stride value is passed unaltered into the appropriate PVM functions.

2.3 Glenda Directed Tuple Operations

In order to improve the overall performance, Glenda offers two operations that send and receive tuples without passing through the tuple server. These two functions are:

- **gl_outto(tid, name,..)** sends a tuple to one or more tasks.
- **gl_into(name,..)** receives a tuple sent directly to the task.

If the first parameter of `gl_outto` is a single integer variable, the tuple is directed to the task identified by this id number. If the first parameter is an integer array, it identifies a collection of tasks to receive the tuple. Since `gl_outto` sends tuples directly from task to task, it is necessary to use `gl_into` for receiving these tuples. Matching within `gl_into` is similar to the other input operations, except that tuples that do not match, are saved within the task instead of within the tuple server.

2.4 Examples

2.4.1 gl_out examples

- `gl_out ( "data", i, k, value )`
  Value can be a scalar or an array. If it is an array, the length is implicit.

- `gl_out ( "row", i, x:len )`
  Here, the length to output for x is given. X could be an array or a pointer.

- `gl_out ( "column", j, x[j]:len )`
  Here, we must specify the length to output. The preprocessor only keeps lengths for single-dimension arrays.

2.4.2 gl_in and gl_inp examples

- `gl_in ( "data", i, k, ? value )`
  `gl_inp ( "data", i, k, ? value )`
  These match on "data", i and k returning data for value. Value can be a scalar or an array. If it is an array, the length is implicit.

- `gl_in ( "row", i, ? x:len )`
  `gl_inp ( "row", i, ? x:len )`
  These match on "row" and i returning data for the array x and returning the number of items as len.

- `gl_in ( "column", j, ? x[j] )`
  `gl_inp ( "column", j, ? x[j] )`
  Assuming that x is a two-dimensional array, these match on "column" and j, returning data into x[j]. The length is ignored.

2.4.3 gl_rd and gl_rdp examples

- `gl_rd ( "data", i, k, ? value )`
  `gl_rdp ( "data", i, k, ? value )`
  These match on "data", i and k returning data for value. Value can be a scalar or an array. If it is an array, the length is implicit.

- `gl_rd ( "row", i, ? x:len )`
  `gl_rdp ( "row", i, ? x:len )`

These match on "row" and i returning data for the array x and returning the number of items as len.

2.4.4 gl_outto and gl_into examples

- `gl_outto ( tid, "data", i, k, value )`
  `gl_into ( "data", i, k, ?  value )`
  In this example, the PVM task id number tid is used to output a tuple containing "data", i, k, and value. Value can be a scalar or an array. The corresponding gl_into matches on "data", i, and k returning the data for value.

- `gl_outto ( tid :  len, "data", i, k, value )`
  `gl_into ( "data", i, k, ?  value )`
  This example is the same as the one above; but instead of being sent to only one process, the tuple is sent to all of the processes whose PVM task id numbers are specified in the array tid of dimension len.

3 Glenda Support Functions

For a variety of reasons, it might be desirable to bypass the Glenda tuple matching operations. The C-Glenda preprocessor provides five operations which pack a collection of PVM function calls into a send/receive facility. The translated code makes use of the functions: `pvm_initsend`, `pvm_send`, and `pvm_mcast`. The support operations are:

- **gl_send(tid, msgtag,...)** sends the package
- **gl_recv(tid,msgtag,...)** receives the package
- **gl_wait(tid,msgtag,...)** receives a message and may wait for more messages.
- **gl_pack** packs the message

- **gl_unpack** unpacks the message.

In the function `gl_send` the first parameter can be a single value or an array of PVM task id numbers, as in `gl_outto`. The second parameter, an integer, represents the PVM message type. The remaining parameters are treated like the components of a tuple.

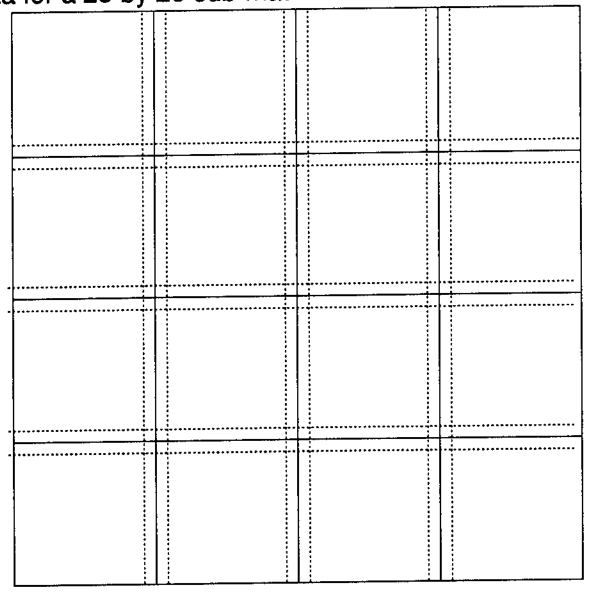
Both `gl_recv` and `gl_wait` receive a message. The operation `gl_recv` receives a message and automatically unpacks it. On the other hand, `gl_wait` simply receives a message and returns the PVM message value. After calling `gl_wait`, either `gl_recv` or `gl_unpack` might be used to unpack the message. In such a way, `gl_wait` can wait for the reception of more than one message (`msgtag =-1`). For completeness, Glenda also offers `gl_pack`, an operation which is compatible with other PVM functions.

4 Glenda and Ocean Circulation Models

This document introduces Glenda, a new parallel processing environment which is modeled after the Linda language and utilizes the PVM software to provide underlying communications. The long-term objectives of this project are to investigate and evaluate the feasibility of parallel processing for ocean circulation applications.

Ocean circulation models usually represent an ocean domain as a 3-D matrix of data, each cell of the matrix consisting of about 5 double precision values. It makes sense to divide the basin into subdomains and allocate one CPU for each subdivision of the original matrix. With this allocation, each CPU computes new values for its matrix and then propagates the boundary values to its neighbors. Then the CPUs would repeat this process of calculating and propagating values until the simulation is completed.

It is clear that the amount of data to be propagated depends upon the coarseness of the grid and the number of CPUs used for the simulation. Let's assume that our grid consists of an N by N square region to be processed by P^2 CPUs. In the figure below we have a 100x100 grid to be processed by 4^2 = 16 CPUs. Each CPU is responsible for computing new data for a 25 by 25 sub-matrix.



In the example, the boundary data is shown with dotted lines. There are 3 ($P - 1$, in general) horizontal divisions with 2 rows of data to be passed (one row goes up and the other goes down). Each row is width 100 (N, in general). This yields a total of $2(P - 1)N$ cells which must be passed up or down in the matrix for one iteration of the simulation. The same is true for the vertical divisions, and the total of all cells to be passed in any direction for one iteration is $4(P - 1)N$.

Let's also assume that there are D bytes of data per cell to propagate after each step of the simulation and that the data transfer rate in the system is R bytes per second. The communication time for one iteration is estimated to be:

$$T = \frac{4(P - 1)ND}{R}$$

For a collection of networked workstations on an Ethernet, R is about 100,000 bytes per second of reliable communication. This is clearly not a large value for R; but networks are commonly busy, so this is a fair estimate.

For a data matrix of about 350x350 horizontal grid points and 20 vertical collocation points, and 5 double precision updating variables, the communication time estimate is:

$$T = \frac{4(P-1)350 \cdot 20 \cdot 5 \cdot 8}{100000} = 11.2 * (P-1)$$

Let's assume that the total CPU time for one iteration is 64 seconds. If new processors are added, the communication time is increased, while reducing the computation time for one iteration. This imposes a limit to the possible speedup in this environment. With our Ethernet example, the limit is reached with 4 processors ($P = 2$). The computation time estimate is about 16 seconds for modern workstations, and the communication time estimate is 11.2 seconds. Adding more CPUs would not improve the efficiency of the system.

It is apparent that Ethernet speed is not a good match for modern CPUs performing ocean modeling. In the case of more tightly-coupled CPUs, as in a multiprocessor machine, the communications are internal and R will be reliably over 10 times as fast. The value $R = 10^7$ bytes per second yields $T = 1.12 * (P-1)$. With 16 CPUs ($P = 4$), the communication time is $T = 3.36$ seconds while the computation time drops to 4 seconds. Using more CPUs would cause the communication time to exceed the computation time.

We realize that for full efficiency in a parallel environment it is necessary to formulate numerical schemes that are most suitable for the new technology. In this respect, implicit schemes, or models formulated with the rigid lid approximation are not optimal. The associated algorithms are connected with the inversion of large matrices, requiring the simultaneous solution of algebraic equations. Explicit schemes or free surface models have a more direct applicability to parallel programming, because their formulation is very regular and requires knowledge of variables at only a few adjacent points.

A work, currently in progress, is configuring an explicit, free surface, barotropic ocean circulation model to parallel processing environments, and verifying the portability of the model, taking advantage of a distributing computing and/or moderately parallel computer system. These findings will be presented in a subsequent document.

Acknowledgments: This work was supported by the Office of Naval Research (under Grant # N00014-92-J-4112).

References

[1] Beguelin, A., et al., *A User's Guide to PVM*, Oak Ridge National Laboratory, Oak Ridge, TN, July 1991.

[2] Carriero, N. and D. Gelernter, 1990: *How to write parallel programs. A first course.* The MIT Press, Cambridge, Mass., 232pp.

[3] Seyfarth, B.R., J.L. Bickham and M.R. Fernandez, *Glenda: An Environment for Easy Parallel Programming*, Scalable High Performance Computing Conference, Knoxville, TN, May 1994.

A Installation and use guidelines

A.1 How to obtain Glenda

To obtain a copy of the Glenda software, e-mail your request to

`seyfarth@whale.st.usm.edu`

and a copy will be sent as soon as possible. In the future, an anonymous `ftp` server may be set up to facilitate the distribution of the software.

A.2 Installation

If you received the `.tar` version, uudecode glenda.tar.Z.uue by typing:

```
> uudecode glenda.tar.Z.uue
```

then decompress glenda.tar by typing:

```
> uncompress glenda.tar.Z
```

and, finally type:

```
> tar -xf glenda.tar
```

If you received the `.shar` version, you have received about 6 e-mail messages which should be saved into separate files. Each file should be edited to remove the header lines as instructed within the files themselves. Then, each file is used as input for the execution of the `sh` command as in:

```
> sh glenda.shar.1
> sh glenda.shar.2
...
```

Either of the distribution methods create Glenda directories within the current directory. The directories created are as follows.

- **glenda/** is the top level directory. It contains the Glenda source code, a Makefile, and additional subdirectories. The subdirectories are:
  - **cgpp/** contains the C-Glenda preprocessor code

- **ts/** contains the tuple server code
- **include/** contains the include files for `ts/` and `examples/` files
- **doc/** contains documentation
- **examples/** contains examples of Glenda programs.

It is essential to specify the architecture of system (ex: ARCH=SUN4) You might edit the Makefile in the `glenda/`, `ts/`, and `examples/` directories to prevent compiling for the wrong architecture.

The Makefiles expect the environmental variable PVM_ROOT to be defined as the root directory of the PVM3.x software and expect to write into the $PVM_ROOT/bin/$ARCH directory.

SGI machines require the linking option `-lsun` to access the XDR routines. This must be added inside the `ts/Makefile`.

This Glenda version is written for PVM 3.x. However, it would need little effort to connect to PVM 2.4. There is a file, `pvmold.c`, which is nearly complete for providing PVM 3.x function calls, using the PVM 2.4 library. Unfortunately, `pvmold.c` does not have an equivalent version of the `pvm_task` function, which is used by the tuple server and `gl_user.c` program to determine the tuple server task id number.

B C-Glenda Preprocessor

The C-Glenda preprocessor converts the source file containing the Glenda functions into a `.c` file, capable of being compiled by any C compiler. The C-Glenda preprocessor is capable also of detecting syntax errors and specifying the type of the errors. Usage is as follows:

```
> cgpp filename.cg
```

(the source code file must end in `.cg`).

B.1 Makefile Sample

The following sample clarifies how to compile the Glenda programs. Make sure that there is a directory `bin/` in the home directory and that the environmental variable ARCH is properly defined.

```
ARCH      =        RS6K
PVMBIN    = $(PVM_ROOT)/bin/$(ARCH)
PVMINCLUDE = -I$(PVM_ROOT)/include -I../include

CC        =        c89

CFLAGS    = -g $(PVMINCLUDE)

PVMLIB    = $(PVM_ROOT)/lib/$(ARCH)
LIB       = -L$(PVMLIB) -lpvm3 -lm

USER      = ../gts/gluser.o

# This part converts a ``.cg'' file to a ``.o'' file.
# The default .SUFFIXES parameter had to be changed to
# accomplish this.
# The -mv command can be removed at your convenience.
.SUFFIXES:
.SUFFIXES: .o .cg .c .f .y .l .s

.cg.o:
        -cgpp $*.cg
        -$(CC) $(CFLAGS) -c $*.c
        -mv $*.c $*.x
```

```
# Place each master file and its corresponding worker file here.

all:    $(PVMBIN)/a $(PVMBIN)/b

$(PVMBIN)/a:        a.o
        $(CC) -o $(PVMBIN)/a a.o $(USER)  $(LIB)
        chmod go+rx $(PVMBIN)/a

$(PVMBIN)/b:        b.o
        $(CC) -o $(PVMBIN)/b b.o $(USER)  $(LIB)
        chmod go+rx $(PVMBIN)/b

clean:
        rm -f *.o
```

C Tuple Server

Before invoking the tuple server, PVM 3.x. must be invoked with the proper configuration parameters. Then, simply type:

```
> gts
```

at the prompt, and the tuple server is automatically placed in the background where it continually tries to receive tuples. At this point, the master process is ready to start.

It is important to not invoke the tuple server without invoking PVM and the master process without invoking the tuple server, first. A typical sequence of commands is as follows:

```
> pvmd pvmhosts &
> gts
> master_filename
```

D Glenda Program Sample

This is an example of a Glenda code. The master file is `a.cg` and the worker file is `b.cg`. In this example, the process, a, outputs an array to multiple processes, b, and waits for each process's response.

a.cg

```
#include <stdio.h>
#include <glenda.h>

main(argc,argv)
int argc;
char *argv[];
{
    int my_tid, a;
    int Size, N, i;
    int *Data;
    int j, Kids;
    int kid, step;

    my_tid = gl_mytid();

    if ( argc > 1 ) Size = atoi(argv[1]);
    else Size = 100000;

    if ( argc > 2 ) N = atoi(argv[2]);
    else N = 10;

    if ( argc > 3 ) Kids = atoi(argv[3]);
```

```
        else Kids = 10;

        gl_out ( "Size", Size );
        gl_out ( "N", N );

        for ( i = 0; i < Kids; i++ ) {
            gl_spawn ( "b" );
            gl_out ( "Kid", i );
        }

        Data = (int *) malloc ( Size * sizeof(int) );

        for ( j = 0; j < N; j++ ) {
            printf("Step %d of %d\n", j+1, N );
            for ( i = 0; i < Kids; i++ )
                gl_out ( "data", i, Data:Size );
            for ( i = 0; i < Kids; i++ ) {
                gl_in  ( "OK", ? kid, ? step );
                printf("Got OK from %d for step %d\n",kid,step+1);
            }
        }

        gl_in ( "Size", Size );
        gl_in ( "N", N );

        gl_exit();
    }
```

b.cg

```
#include <stdio.h>
#include <glenda.h>

main(argc,argv)
int argc;
char *argv[];
{
    int my_tid, a;
    int Size, N, i;
    int *Data;
    int k;

    my_tid = gl_mytid();

    gl_rd ( "Size", ? Size );
    gl_rd ( "N", ? N );
    gl_in ( "Kid", ? k );

    fprintf(stderr,"Kid %d, Size %d, N %d\n",k,Size,N);

    Data = (int *) malloc ( Size * sizeof(int) );

    for ( i = 0; i < N; i++ ) {
        gl_in ( "data", k, ? Data:Size );
        gl_out ( "OK", k, i );
    }

    gl_exit();
}
```

The files `a.cg` and `b.cg` are located in the directory `glenda/examples/`. Another example includes the files `mm.c` and `mmworker.c` which are the master and worker programs, written in PVM, for the execution of matrix multplications. The files, created by Josef Fritscher (Technical University of Vienna) were acquired from the newsgroup `comp.parallel.pvm`.

The files `mmgl.cg` and `mmgl_worker.cg` are the corresponding Glenda versions. The files `mmto.cg` and `mmto_worker.cg` are also Glenda versions of `mm.c` and `mmworker.c`, but they make use of the `gl_outto` and `gl_into` operations.

E Getting Help

Please communicate problems and bug reports to

`seyfarth@whale.st.usm.edu`

It would be helpful to describe your virtual machine configuration (hardware, PVM version), include a short segment of code illustrating the problem, and describe how it fails.

Good luck with your work !

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. Agency Use Only *(Leave blank).*	**2. Report Date.** February 1995	**3. Report Type and Dates Covered.** Technical Report

4. Title and Subtitle.

GLENDA SOFTWARE DESIGN

5. Funding Numbers.

Program Element No.

Project No.

Task No.

Accession No.

6. Author(s).

Benjamin R. Seyfarth
Jerry L. Bickham
Germana Peggion

7. Performing Organization Name(s) and Address(es).

Center for Ocean & Atmospheric Modeling
The University of Southern Mississippi
Building 1103, Room 249
Stennis Space Center, MS 39529-5005

8. Performing Organization Report Number.

TR-3/95

9. Sponsoring/Monitoring Agency Name(s) and Address(es).

Office of Naval Research
Code 1513: RKL
Ballston Centre Tower One
800 North Quincy Street
Arlington, VA 22217-5660

10. Sponsoring/Monitoring Agency Report Number.

11. Supplementary Notes.

ONR Research Grant No. N00014-92-J-4112

12a. Distribution/Availability Statement.

Approved for public release; distribution is unlimited.

12b. Distribution Code.

13. Abstract *(Maximum 200 words).*

Glenda is an environment for parallel processing which is modeled after the Linda language and utilizes the PVM software system to provide underlying communications. The resulting software maintains the friendly parallel programming, typical of Linda, and the PVM efficiency in message-passing operations.

This document describes the functions, data structure, and algorithms of the Glenda software architecture.

14. Subject Terms.

(U) GLENDA, (U) LINDA, (U) PVM, (U) PARALLEL, (U) NETWORK, (U) PROGRAMMING, (U) TUPLE, (U) PRE-PROCESSOR

15. Number of Pages. 25

16. Price Code.

17. Security Classification of Report. Unclassified	**18. Security Classification of This Page.** Unclassified	**19. Security Classification of Abstract.** Unclassified	**20. Limitation of Abstract.** SAR